**OCL: Object Constraint Language**

**Giannis Tzitzikas**

**Διάλεξη:** 14
**Ημερομηνία:**

---

**Διάρθρωση**

- Στόχοι της OCL
- Γιατί να χρησιμοποιήσουμε την OCL
- Παρουσίαση της OCL
- Διαβεβαιώσεις και Γλώσσες Προγραμματισμού
  - (Assertions and Programming Languages)
Τι είναι η OCL (Object Constraint Language)?

- Μια τυπική γλώσσα (formal language) για την προδιαγραφή περιορισμών (constraints) σε αντικειμενοστρεφή μοντέλα
- Είναι δηλωτική (declarative) (περιγράφει το τι αντί του πως)
- Είναι μια γλώσσα με τύπους (typed)
  - Και πιο φιλική από άλλες τυπικές γλώσσες

Περί περιορισμών

- Κάποιοι περιορισμοί μπορούν να εκφραστούν γραφικά με τη γραφική γλώσσα UML (π.χ. η πολλαπλότητα των συσχετίσεων, partition subclasses, κλπ).
- Για κάποιους άλλους αυτό δεν είναι εύκολο/εφικτό, π.χ.:
  - Περιορισμοί που εμπλέκουν >2 κλάσεις
  - Περιορισμοί που εμπλέκουν τιμές γνωρισμάτων (και συνδυασμούς αυτών)
  - Προϋποθέσεις και Μετα-συνθήκες (pre/post-conditions) λειτουργιών

Η OCL μπορεί να τους εκφράσει με τυπικό τρόπο

Γιατί να γράφουμε περιορισμούς σε OCL;

Why to write OCL constraints?

Γιατί να εκφράσουμε ηρήμως τέτοιους περιορισμούς;
Γιατί κάνουν τα μοντέλα μας πιο ακριβή

- Ωστε να τα καταλαβαίνουμε καλύτερα
- Ωστε οι προγραμματιστές να τα υλοποιούν (σωστά)
- Ωστε να μπορούμε να έχουμε μια τυπική επαλήθευση (formal validation) του μοντέλου πριν την υλοποίηση
  - και δοκιμασίες (tests) για τη φάση της υλοποίησης

Μπορούν να μεταφραστούν σε «διαβεβαιώσεις» (assertions) στις γλώσσες προγραμματισμού
- Μερικά εργαλεία CASE προσφέρουν τέτοιες μεταφραστικές και επαληθευτικές υπηρεσίες
Τα διαγράμματα κλάσεων δεν είναι πολύ ακριβή

- Μπορεί ένας ανήλικος να εργαστεί σε μια εταιρία;
- Μπορεί μια εταιρία να προσλάβει ένα άτομο που είναι ήδη εργαζόμενος της;

Το παραπάνω διάγραμμα δεν μας αποτρέπει τίποτα από τα παραπάνω

- Μπορεί ένα πρόσωπο να αρχίσει να εργάζεται πριν τη γέννησή του;
- Μπορεί μια προαγωγή να μειώσει το μισθό ενός εργαζομένου;
- Υπάρχει κάποιο κατώτερο όριο στους μισθούς για αυτούς που εργάζονται στην εταιρία πάνω από 10 έτη;
Τα διαγράμματα κλάσεων δεν είναι πολύ ακριβή (III)

...Η σπουδαιότητα της γνώσης υποβάθρου

• Θα μπορούσατε να αναπτύξετε ένα σύστημα (στον οποίο να είχατε λάβει υπόψη τους προηγούμενους περιορισμούς) αν τα διαγράμματα κλάσεων ήταν σε μια γλώσσα που δεν γνωρίζατε (π.χ. στα Ισπανικά);

Τα διαγράμματα κλάσεων δεν είναι πολύ ακριβή (IV)
Object Constraint Language (OCL)

- OCL is a formal language used to describe expressions on UML models.
- OCL expressions typically specify invariant conditions that must hold for the system being modeled.
- They can also specify queries over objects described in a model.
- OCL is a typed language, so that each OCL expression has a type. To be well-formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String.
- When OCL expressions are evaluated, they do not have side effects; i.e., their evaluation cannot alter the state of the corresponding executing system.
  - However, OCL expressions could be used to specify operations / actions that, when executed, do alter the state of the system.

Πότε τη χρησιμοποιούμε;
Where to use OCL?

OCL can be used for a number of different purposes:
- To specify invariants on classes and types in the class model
- To describe pre- and post conditions on Operations and Methods
- To specify derivation rules for attributes for any expression over a UML model.
- To describe Guards in State Diagrams
- To specify target (sets) for messages and actions
- To specify type invariant for Stereotypes
- As a query language

UML modelers can use OCL to
- to specify application-specific constraints in their models.
- to specify queries on the UML model, which are completely programming language independent
The main types of OCL Constraints

- **Invariants on classes** (αναλλοίωτες συνθήκες στις κλάσεις)
  - συνθήκες που πρέπει να ικανοποιούνται από όλα τα στιγμιότυπα μιας κλάσης
    - Π.χ., salary > 1000 Euro

- **pre-conditions on operations** (προ-συνθήκες στις λειτουργίες)
  - συνθήκες που πρέπει να ικανοποιούνται πριν την εκτέλεση μιας λειτουργίας
    - Π.χ., η λειτουργία «Απόλυση()» μπορεί να εκτελεστεί μόνο σε έναν άτομο που έχει ήδη προσληφθεί

- **post-conditions on operations** (μετα-συνθήκες στις λειτουργίες)
  - συνθήκες που πρέπει να ικανοποιούνται μετά την εκτέλεση μιας λειτουργίας
    - Π.χ., μετά την εκτέλεση της «Ανάληψη(ποσό)» το υπόλοιπο του τραπεζικού λογαριασμού πρέπει να έχει μειωθεί κατά «ποσό».

How we can specify a constraint?

- Declaration of the **context** of a constraint by referencing the model element that a constraint applies to

- Declaration of the **type** of a constraint (**inv**, **pre**, **post**)

- Expressing the desired condition by referencing properties of model elements and using various operations that are supported.
Δήλωση Συμφραζομένων (Συγκειμένων)

Context Declaration

- Προσδιορίζει το στοιχείο στο οποίο αφορά ο περιορισμός
- Το context μπορεί να είναι
  - a class (for invariants)
  - an operation (for pre/post-conditions)

• Παράδειγμα:

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>age</td>
</tr>
<tr>
<td>salary</td>
</tr>
<tr>
<td>SetAge(a)</td>
</tr>
<tr>
<td>SetSalary(s)</td>
</tr>
</tbody>
</table>

| Context Employee inv: self.salary > 1000 |

| Context Employee::SetSalary(salary) pre: salary > 1000 |

Δεν είναι ισοδύναμα. Γιατί;

Ονόματα και σχόλια περιορισμών

Constraint names and comments

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>age</td>
</tr>
<tr>
<td>salary</td>
</tr>
<tr>
<td>SetAge(a)</td>
</tr>
<tr>
<td>SetSalary(s)</td>
</tr>
</tbody>
</table>

| Context Employee::SetAge (age) pre: age > 0 |

| Context Employee::SetAge (age) pre positive_age : age > 0 |

Optional constraint name
Allowing the constraint to be referenced by name.

| Context Employee::SetAge (age) pre positive_age : age > 0 |

-- the age should always be positive

Comment
Everything immediately following the two dashes up to and including the end of line is part of the comment.
In most cases, the keyword `self` can be dropped because the context is clear. As an alternative for `self`, a different name can be defined playing the part of `self`.

\[
\text{Context } e: \text{Employee} \\
\text{inv: } e.\text{salary} > 1000
\]

\[
\text{Context Employee} \\
\text{inv: } \text{self}.\text{salary} > 1000
\]

---

**Selectors (how we reference elements)**

<table>
<thead>
<tr>
<th>Person</th>
<th>0..* employment</th>
<th>0..1 Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>employee</td>
<td>name</td>
</tr>
<tr>
<td>age</td>
<td>president</td>
<td>stockPrice()</td>
</tr>
</tbody>
</table>

\[
\text{self.age} \quad // \text{returns the age of a particular person}
\]

\[
\text{self.employment} \quad // \text{returns the employer (company) of a person}
\]

\[
\text{self.employer} \quad // \text{as before}
\]

\[
\text{self.employment} \quad // \text{returns the set of all employees of a company}
\]

\[
\text{self.president} \quad // \text{returns the singleton with the president of a company}
\]

\[
\text{self.stockPrice()} \quad // \text{returns the value this method would return}
\]
Selectors (how we reference elements)

Because the multiplicity of the role manager is one, `self.manager` is an object of type `Person`. This happens when the multiplicity of the association-end has a maximum of one ("0..1" or "1")

Selectors
Referencing Association Classes

The salary should be $> 1000$

We use dot and the name of the association class starting with a lowercase letter
The age of a children should be less than the age of its parents.

Here, the name of the association class alone is not enough. We need to distinguish the direction in which the association is navigated.

To make the distinction, the rolename of the direction in which we want to navigate is added to the association class name, enclosed in square brackets.

Context Person inv: self.hasParent[parent].age > self.age

self.hasParent.age is invalid

Let c be a company. The name of the company that owns c should be different than the name of c.

Context Company inv: self.owns[owner].name <> self.name
• **Boolean Operations**
  - **and** // $\wedge$
  - **or** // $\vee$
  - **not** // $\neg$
  - **implies** // $\rightarrow$
  - **xor**

• **Comparison operations**
  - $<$, $>$, $<=$, $>=$, $<=$, $==$ 

• **Arithmetic**
  - $+, -, *, /, \text{abs}(), \text{div}, \text{floor}(), \text{round}()$

• **String operations**
  - $\text{concat}(s1, s2)$, $\text{toUpper}(s)$,
  
• **Nil**
  - if an attribute $\text{attr}$ of an object $\text{obj}$ has no value then $\text{obj}.\text{attr}$ returns $\text{nil}$

• **Empty**
  - if there are no associated objects to an object $\text{obj}$ through an association $\text{assoc}$ then $\text{obj}.\text{assoc}$ returns the empty bag $\{}$.

• **Nil <> Empty**

---

**Referring to enumerations**

• **Enumerations**

```
Person
<table>
<thead>
<tr>
<th>salary</th>
<th>sex</th>
</tr>
</thead>
</table>

JobType
+ admin
+ programmer
+ secretary

Context Person inv: self.job==JobType::admin implies self.salary > 10.000

Context Person inv: self.name=="Yannis" implies self.sex::Male
```

---
• Allow us to refer to the objects that are referred using associations (typically in those with upper multiplicity > 1)

<table>
<thead>
<tr>
<th>Person</th>
<th>age</th>
<th>works</th>
<th>Company</th>
<th>name</th>
</tr>
</thead>
</table>

p1: Person
p2: Person
c1: Company
c2: Company
c3: Company

c2.works
p1.works

Collections in OCL (II)

Single navigation of an association results in a **Set**, combined navigations in a **Bag**, and navigation over associations adorned with {ordered} results in an **OrderedSet**.

Collection is an abstract type, with the concrete collection types (Set, Sequence, and Bag) as its subtypes.
Objects
- are instances of classes, including the predefined ones (e.g. Integer)

Sets
- a “set” of objects
- example: Set { p1, p2}

Bag
- duplicates allowed
- example: Bag { p1, p1, p1, p2, p1}

Sequence
- is a bag of ordered elements
- example: Sequence {p1, p2, p3, p1} // <p1, p2, p3, p1>

Collection Operations

- c1 -> Size() // number of elements of c1
- c1 -> count(elem) // counts the number of occurrences of elem in c1
- c1 -> includes(elem) // checks if elem is member of c1
- c1 -> includesAll(coll) // checks if coll is contained in c1
- c1 -> excludes(elem) // returns True if elem is not member of in c1
- c1 -> isEmpty() // checks if c1=()

- c1 -> forAll(expr) // returns True if expr is true for all elements of c1
- c1 -> exists(expr) // returns True if expr is true for at least one element of c1
- c1 -> select(expr) // returns the elements of c1 that satisfy expr
- c1 -> reject(expr) // returns the elements of c1 that do not satisfy expr

SET OPERATIONS:
- c1 -> union(c2), c1 -> intersection(c2), c1-c2
**Collection Operations: Examples**

- `p1.works->size()` is 2
- `p1.works->count(c3)` is 0
- `p1.works->includes(c2)` is True
- `p1.works->includes(c3)` is False
- `c2.works->includesAll(c1.works)` is True
- `c2.works->includesAll(c2.works)` is False
- `c3.works->isEmpty()` is True

**Collection Operations: Examples (II)**

- `c2.works->forall( x | x.age>20 and x.age < 70)` is False
- `c2.works->exists( x | x.age>20 and x.age < 70)` is True
- `c2.works->select( x | x.age>20 and x.age < 70)` will return {p2}
- `p1.works->intersection(p2.works)` will return {c2}
- `p1.works - p2.works` will return {c1}`
A single object can be used as a Set as well. It then behaves as if it is a Set containing the single object. The usage as a set is done through the arrow followed by a property of Set.

**context** Company

**inv:** self.manager->size() = 1

---

**Select / Reject**

The reject operation is available in OCL for convenience, because each reject can be restated as a select with the negated expression. Therefore, the following two expressions are identical:

\[
\text{collection->reject( } v : \text{Type | boolean-expression-with-v )}
\]

\[
\text{collection->select( } v : \text{Type | not (boolean-expression-with-v) )}
\]

The collection of all the employees who are not married is empty:

**context** Company

**inv:** self.employee->reject( isMarried )->isEmpty()
Collect operation

The select and reject operations always result in a sub-collection of the original collection.

When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a collect operation.

The collect operation uses the same syntax as the select and reject and is written as one of:

- `collection->collect( v : Type | expression-with-v )`
- `collection->collect( v | expression-with-v )`
- `collection->collect( expression )`

The value of the reject operation is the collection of the results of all the evaluations of `expression-with-v`.

An example: specify the collection of birthDates for all employees in the context of a company. This can be written in the context of a Company object as one of:

- `self.employee->collect( birthDate )`
- `self.employee->collect( person | person.birthDate )`
- `self.employee->collect( person : Person | person.birthDate )`

Collect (2)

Shorthand for Collect

Because navigation through many objects is very common, there is a shorthand notation for the collect that makes the OCL expressions more readable.

Instead of

- `self.employee->collect(birthdate)`

we can also write:

- `self.employee.birthdate`

In general, when we apply a property to a collection of Objects, then it will automatically be interpreted as a collect over the members of the collection with the specified property. For any property name that is defined as a property on the objects in a collection, the following two expressions are identical:

- `collection.propertyname`
- `collection->collect(propertyname)`

and so are these if the property is parameterized:

- `collection.propertyname (par1, par2, ...)`
- `collection->collect (propertyname(par1, par2, ...))`
Collect (3)

When the source collection is a **Set** the resulting collection is not a Set but a **Bag**.

If the source collection is a **Sequence** or an **OrderedSet**, the resulting collection is a **Sequence**.

When more than one employee has the same value for birthDate, this value will be an element of the resulting **Bag** more than once.

The **Bag** resulting from the collect operation always has the same size as the original collection.

It is possible to make a **Set** from the **Bag**, by using the **asSet** property on the **Bag**.

Example:

```
self.employees->collect(birthDate)->asSet()
```

Results in the **Set** of different birthDates from all employees of a **Company**

---

Examples with Bags and other operations

<table>
<thead>
<tr>
<th>Person</th>
<th>employment</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>*</td>
<td>name</td>
</tr>
<tr>
<td>age</td>
<td></td>
<td></td>
</tr>
<tr>
<td>income</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hire(c, date)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fire(c, date)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>increaseAge()</td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>promote(inc)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **employment.age** is a **bag**
- **employment.income** is a **bag**
- **employment.income->asSet()** returns all distinct incomes of the employees
Examples of Invariants (using collection operations)

All persons should have positive age

Context Person inv: self.age > 0

All persons that work for a company should be adults

Context Company inv: self.employment->forall( x | x.age > 18)

A person can be a manager of only one company

Context Company inv: not (self.manager->exists(x) x.employer->exists(y|y<>self))

Context Company inv: self.manager.employer->forall(x | x = self)

All companies should have managers that are not employers of other companies
Another example

Context Person

inv: self.parent->excludes(self) and self.children->excludes(self)

Forall

context Company

inv: self.employee->forAll( age <= 65 )
inv: self.employee->forAll( p | p.age <= 65 )
inv: self.employee->forAll( p : Person | p.age <= 65 )

These invariants evaluate to true if the age property of each employee is less or equal to 65.

The forAll operation has an extended variant in which more then one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a forAll on the Cartesian product of the collection with itself.

context Company inv:
self.employee->forAll( e1, e2 : Person | e1 <> e2 implies e1.forename <> e2.forename)

This expression evaluates to true if the forenames of all employees are different. It is semantically equivalent to:

context Company inv:
self.employee->forAll( e1 | self.employee->forAll( e2 | e1 <> e2 implies e1.forename <> e2.forename))
Examples of Constraints (using collection operations)

Pre/Post-Conditions

<table>
<thead>
<tr>
<th>Person</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
</tr>
<tr>
<td>age</td>
<td></td>
</tr>
<tr>
<td>income</td>
<td></td>
</tr>
</tbody>
</table>

- hire(c, date)
- fire(c, date)
- increaseAge()
- promote(inc)

Context Person::hire(c:Company)

pre: not employment->includes(c)
post: employment->includes(c)

Context Person::fire(c:Company)

pre: employment->includes(c)
post: not employment->includes(c)

Context Person::increaseAge()

post: age = age@pre + 1

Context Person::Promote (inc)

post: self.income = income@pre * (1+inc)

@Pre

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object.

- a.b@pre.c -- takes the old value of property b of a, say object18, and then the new value of c of object18.
- a.b@pre.c@pre -- takes the old value of property b of a, say object18, and then the old value of c of object18.

The ‘@pre’ postfix is allowed only in OCL expressions that are part of a Postcondition.

Asking for a current property of an object that has been destroyed during execution of the operation results in OclUndefined. Also, referring to the previous value of an object that has been created during execution of the operation results in OclUndefined.
Post-conditions

Result, out-parameters

The reserved word \texttt{result} denotes the result of the operation, if there is one.

\begin{verbatim}
Context Person::getIncome(d:Date): Integer
  post: result = 1000
\end{verbatim}

The right-hand-side of this definition may refer to the operation being defined (i.e., the definition may be recursive) as long as the recursion is not infinite.

When the operation has no \texttt{out} or \texttt{in/out} parameters (like in this example), then the type of result is the return type of the operation (here \texttt{Integer}).

When the operation has \texttt{out} or \texttt{in/out} parameters, the return type is a \texttt{Tuple}.

The postcondition for the income operation with an \texttt{out} parameter \texttt{bonus} could be:

\begin{verbatim}
Context Person::getIncome(d:Date, bonus:Integer): Integer
  post: result = Tuple\{bonus=300, result=1000\}
\end{verbatim}

The return type of operation calls is \texttt{Tuple( bonus: Integer, result: Integer)}.

The out parameters need not be included in the operation call (we have to provide values only for the \texttt{in} or \texttt{in/out} parameters).

Let \texttt{Yannis} be an object of the class \texttt{Person}, and let \texttt{d1} be a \texttt{Date}. Then, \texttt{Yannis.getIncome(d1)} is a valid operation call.

The type of the result of this operation call is \texttt{Tuple( bonus: Integer, result: Integer)}.

We can access these values using the names of the out parameters, and the keyword result, for example:

\texttt{Yannis.getIncome(d1).bonus = 300 and}
\texttt{Yannis. getIncome(d1).result = 1000}
**Body:** Indicating the result of a query operation

An OCL expression may be used to indicate the result of a query operation.

The expression must conform to the result type of the operation. Like in the pre/post-conditions, the parameters may be used in the expression. Pre/post-conditions, and body expressions may be mixed together after one operation context.

**Context**

```
Person::getCompany():Company
pre: self.employment->size()>0
body: self.employment
```

**Body:** Indicating the result of a query operation (II)

**Context**

```
Person::getCurrentSpouse():Person
pre: self.isMarried = true
body: self.marriage ->select( m| m.ended = false).spouse
```
(to be continued)